1995

Software Architectures in Computer-Aided Prototyping

Luqi

Software Architectures in Computer-Aided Prototyping *

Luqi, Valdis Berzins
Computer Science Department
Naval Postgraduate School, Monterey, CA 93943, USA

Abstract

We examine representations and support for software architectures in the context of computer aided prototyping. To assess the potential contributions of advances in this area, we explore the connection between generic software architectures and automation support for software reuse, program generation, software evolution, reengineering, and transformation of prototypes into production code.

1 Introduction

The study of software architecture is concerned with the large scale structure and design of software systems. A software architecture defines the common structure of a family of systems by identifying and specifying (1) the components that comprise systems in the family, (2) the relationships and interactions between the components, and (3) the rationale for the design decisions embodied in this information. The concept of software architecture was generalized to cover the structure of a family of systems rather than just a single system to better support software evolution and reuse. Software architecture is a relatively new field of study, and many variations on this basic idea have been proposed [1].

Languages are being developed to describe software architectures. An architecture language has to be based on a model of software architectures, and the properties of that model have a strong influence on the complexity and usefulness of the language. Designing a general purpose architecture language is very difficult because there is such a wide range of architectural possibilities, especially if all details are included. This makes it very hard to cover the subject without making the language so complicated that it becomes difficult to use. The problem is somewhat easier if the application domain is narrowed to address only one type of application. For example, the types of architectures used in business information systems are different from those used in embedded real-time systems.

This paper examines software architectures in the context of computer-aided prototyping, explains the architecture models used in that context, presents some examples of representations

*This research was supported in part by the National Science Foundation under grant number CCR-9058453 and in part by the Army Research Office under grant number ARO 111-95.

for software architectures, and outlines some of the associated automation support. We also indicate some extensions to these representations and relate our results to software architectures in a more general context.

2 Related Work

Software architectures are receiving increasing attention because they are closely related to many aspects of software development:

Synthesis. Large and complex systems are realized by interconnecting components in hierarchical assemblies. Thus support for designing interconnections of subsystems is a key to rapid and economical construction of large systems [8].

Understanding. People can understand complex designs only by organizing them into levels that have relatively small and simple realizations in terms of the components at the next level down. Since understanding is a prerequisite for quality control, the choice of software architecture strongly influences the reliability and appropriateness of software products.

Evolution. It is necessary to understand the structure of a system and its principles of operation in order to plan and reliably adapt the design to a requirements change. Since the structure of the system has a strong influence on the cost of a change, the well known principle of information hiding [10] urges system architects to confine decisions that are likely to change within individual components. It is quite difficult to anticipate what decisions will change, so that the right software architecture is often found only after several attempts at designing systems in a given domain. Improvements in support for designing and modifying software architectures can increase software flexibility [5].

Reuse. The reusability of both components and interconnection structures is critically dependent on architectural coherence. Experience has shown that prior planning and deliberate design for multiple use are needed for effective reuse. Arbitrary pieces of code are usually not reusable. Reusable components are designed to fit into architectures that cover a large span of applications, and reusable architectures are consistent with many well-defined options for component behavior.

Integration. Different systems cannot work together effectively unless their interfaces are consistent. Software architectures define the standards, conventions, and common conceptual models needed to achieve such consistency [3]. Thus agreement on a common architecture at the highest levels is a prerequisite for interoperability of systems developed by different organizations.

Analysis. Explicit representations of architectural information enable new forms of computer aided software analysis and decision support for system designers, such as consistency checking [3] and real-time scheduling [8]. The localized structure that enables human understanding also appears to be needed for automated decision support, because the computational requirements for many analysis tasks increase sharply with the number of components to be considered.

Management. The structure of the software architecture is closely related to the structure of the human activities needed to construct or modify the software system. Explicit representations for software architectures thus enable decision support for project planning and completely automated scheduling, work assignments, and configuration management [2]. The design rationale aspect of software architectures enables decision support for planning software evolution efforts [11].

Current best commercial practice with respect to software architectures appears to center on toolkits and frameworks with limited automated decision support [9]. Experimental languages for describing software architectures and composing systems from large-scale components (megaprogramming) have been proposed.

The megaprogramming language MPL considers the problem from a database perspective, and considers issues such as data transformations required when crossing subsystem boundaries, optimization of large scale actions, and dynamic monitoring of progress to control scheduling and execution strategies [13]. The architecture language UniCon specifies both software functionality as well as packaging properties of software components and connectors [1]. The system architecture language Rapide is aimed at distributed systems, captures dynamic as well as static connection patterns, and provides simulation and behavior analysis functions [1]. These languages address complex software products, and strive for comprehensive coverage of architectural properties. The languages themselves are also fairly complicated.

Experimental systems are also emerging for automatically composing programs for solving problems over a fixed problem domain based on a given architecture and a set of components consistent with that architecture. These systems take a somewhat different approach: the architectural information is not intended for human consumption, but rather is used internally by software tools that automatically create instances of the architecture that realize particular applications. Some of the systems in this category include AMPHION, a system for constructing programs that do astronomical calculations from graphical descriptions of the situations to be analyzed [7], Panel, a system for constructing multimedia animations [7], SDDR, a system for creating reliable and reusable software designs [7], ControlH and MetaH, systems for developing control software [7], and CAPS, a system for creating prototypes of real-time systems [8].

Our interest in software architectures is motivated by the desire to provide computer aid for the software prototyping process [6]. Iterative prototyping is characterized by repeated and substantial changes that focus on a common theme determined by the known and unknown needs of the clients and the areas of the greatest uncertainty. The requirements and a software architecture for realizing those requirements are developed concurrently via an iterative process that uses prototype demonstration to elicit adjustments to requirements. The software architecture is developed as a necessary by product, which is required to realize the executable version of the prototype. The software architecture serves a dual role in this process, because it serves both as the design for the initial version and as a description of a family of similar systems. Each step in the prototyping process can be viewed as navigation in this family of systems, with the goal of moving from the current point to one closer to the needs of the clients.

The focus of our work has been to support rapid prototyping and large scale software design, particularly for large, distributed, and real-time systems. The prototyping language PSDL associated with the CAPS system is an early architecture language tailored to support automated

generation of connections, automated real-time scheduling, computer-aided software reuse, and computer-aided software evolution [8]. PSDL is coupled with a variety of formal notations for describing behavior of individual software components, such as the specification language Spec [3].

CAPS and PSDL were developed before software architectures emerged as an independent subject. Although the purpose of the CAPS project was to provide computer aid for prototyping rather than developing deliverable software, the effort addressed many problems related to software architectures. The rest of this paper summarizes these results and suggests some extensions that contribute to software architectures in a wider context.

3 Representing Architectures in Prototyping

Prototyping requires rapid realization and analysis of proposed system behaviors, so that designers can iteratively approach an accurate formulation of client needs and corresponding software solutions. PSDL was designed to achieve the required speed through *simplicity*.

This goal was achieved by introducing architectural abstractions to eliminate as many details from the designer's consideration as possible, to separate the specification of the essential aspects of the abstract architecture from implementation aspects of the concrete architecture, and to minimize the implementation aspects imposed on the designer by automating the choice of as many implementation details as possible.

The result is a spartan representation for abstract system architectures, summarized in Fig. 1. A PSDL architecture is a connection pattern that specifies how a set of components are to interact by defining connections and constraints. The rationale for the structure is represented via links to system requirements together with formal and/or informal descriptions of intended component behavior. This information is organized in a hierarchy. Generalization is supported by generic components and connections. The rest of this section describes these aspects of the architecture model underlying the PSDL representation in more detail.

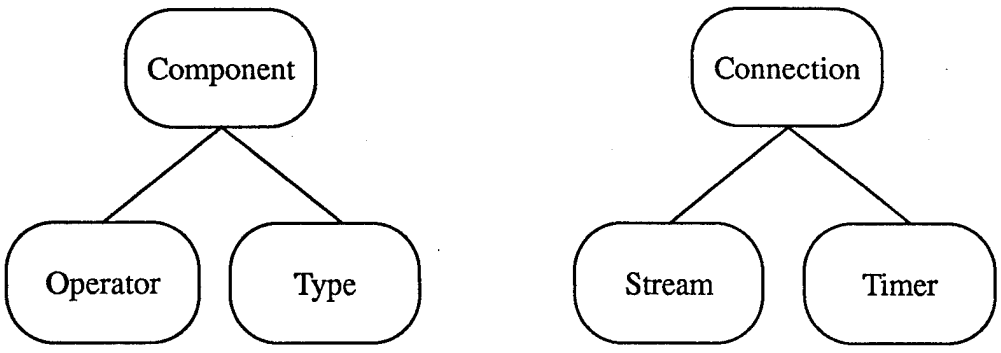

Figure 1: PSDL Component and Connection Types.

3.1 Interactions

PSDL has just two generalized types of interactions between systems: data streams and timers. This model is the simplest possible in the sense that these types of interactions are orthogonal: the first is mediated by transmission of data and the second is mediated by the passage of time. Experience has shown that they are sufficient for describing a variety of embedded real-time systems.

PSDL models connection patterns for operators as annotated networks of subordinate operators communicating via data streams. This model can be represented mathematically as an augmented directed hypergraph whose nodes are operators and whose edges are streams. Edges in a directed hypergraph can have multiple sources (operators writing into the stream) and multiple sinks (operators reading from the stream). This structure can be displayed graphically in a format similar to traditional data flow diagrams.

A simple computational model is associated with this structure. When an operator fires, it reads one data value from each of its input and state streams, and writes at most one data value into each of its output and state streams.

The hypergraph representing the connection pattern is annotated with timing and control constraints attached to the operators and streams. The timing and control constraints determine the conditions under which the operators are activated (i.e. can be fired).

The main simplification realized in the component interaction model is due to a general abstract model of system interactions that hides details of protocols and unifies data and control flow into a single type of interaction, the data stream. Data streams are generalized objects that have specializations with respect to several dimensions, including the data type whose values are carried by the stream, whether the stream represents a state variable, whether the stream models a discrete or continuous data source, whether the stream connects different processors, and the implementation languages of the producer and consumer systems. The data type carried by a stream is part of the abstract architecture of the system and must be specified explicitly by the designer.

The other properties are hidden from view because they are automatically derived from the abstract and implementation properties of the subsystems that interact via the stream. This simplifies the designer's view and removes an opportunity for introducing inconsistencies into the architecture. This represents a radical departure from other approaches to architecture. Rather than treating a component as an individual and specific piece of code that has a specific and detailed communication protocol, we consider the component as a module with abstract input and output patterns that can be realized in a variety of concrete protocols as the need arises, via generated code. At a high level of design, packaging aspects are irrelevant. If connections can be automatically realized, then packaging aspects become largely irrelevant for concrete realizations as well, at least in the context of prototyping.

Concrete packaging can impact optimizations that may have to be performed to transform prototypes into product-quality implementations. High level representations for this kind of information have been explored in the context of the Spec language, where they are represented via optional refinement declarations that define implementation strategies [3]. This structure is a high level analog to pragmas in Ada, except that the declarations represent binding constraints rather than optional implementation advice. Treating component interactions abstractly and providing

automated support for realizing and choosing concrete packaging aspects of component interactions is attractive for production code as well as for prototyping because it eases performance tuning of large systems.

Communication via side effects on shared mutable objects is excluded from the abstract architecture in the context of prototyping because such constructions can easily introduce faults due to unplanned interactions. Preventing this failure mode speeds up prototype realization and evolution by reducing debugging time.

This prohibition does not exclude concrete implementations that communicate via side effects. It does confine these mechanisms to optimizations introduced in the automatically generated realizations of streams. Such optimizations are strictly implementation level details that have no effect on the abstract architecture: the connection generator has an obligation to ensure that the behavior of optimized realizations conforms exactly to the behavior of the abstract architecture. This supports one of the central principles of large scale software design: subsystems must interact *only* via the specified interfaces.

The connection model of PSDL also supports another basic principle of large scale software design, which says that the specification and implementation of a component should be independent of the context in which it is used. The interface to an operation refers only to the input streams and output streams of the operation. Both the specification and the implementation of each operation are completely independent of where the data in the input streams comes from and where the data in the output streams goes. Thus an operation can be connected into compatible slots in many different architectures without any fear that this might inadvertently change the behavior of the component in some way. This enhances reusability and flexibility of both components and connection patterns.

PSDL uses a variety of notations for defining the required behavior of components, including Spec. Similarly to PSDL, Spec was designed to provide a unifying abstraction for interactions between subsystems. The spec model of interactions is based on the event model [3], in which data items are associated with events caused by data transmissions. This abstracts from differences in realizations of control and data connections, such as subprogram calls vs. rendezvous vs. gotos and parameter passing vs. I/O vs. global data. The Spec model of interactions is more restricted than the PSDL model. In Spec all of the data components associated with an event come from the same source, while in PSDL the input streams of an operator can come from different sources. This difference is not significant when both notations are used together because Spec can be used within PSDL only to define the behavioral requirements for an individual component slot in the architecture.

3.2 Components

The PSDL component model is also very simple: all components are either data types or operators (logical processes). Abstract architectural aspects of components are separated from incidental aspects of concrete realizations. Implementation considerations such as packaging into physical processes, mutual exclusion, locking, synchronization, and flow of control are all automatically realized by generated code, based on declared control and timing constraints associated with the operators.

At the logical level, all data types occurring in a PSDL architecture are immutable. At the

level of concrete realizations, mutable representations are allowed as optimizations provided that they correspond exactly to the specified abstract behavior.

All states in the abstract architecture are associated with data streams or timers representing state variables. Every state variable is local to some operator component, although its current value can be transmitted on output streams. This realizes the strict encapsulation required for independent modules.

3.3 Constraints

PSDL connection graphs are augmented with constraints that are associated with component slots in the architecture. These constraints serve several purposes:

- Some control constraints define the conditions under which each component can be activated. These act as execution guards and document the assumptions that can be made in the implementation of the component that fits into the architecture slot.
- Other control constraints limit or augment the runtime behavior of the components. These constraints serve to make small adjustments to the behavior of existing components. This capability enables reuse of components that do not quite match the designer's needs and supports evolution of systems containing legacy code whose source is not available for modification. Examples are output guards that suppress component outputs that do not satisfy specified conditions, exception constraints that raise exceptions if outputs do not satisfy specified conditions, and timer constraints that can start, stop, and reset timers under specified conditions.
- Timing constraints specify how often and how quickly operations must be performed. These constraints identify the time-critical aspects of the system and determine how computing resources must be allocated to meet the timing requirements associated with interactions between components.

The control constraints are realized by automatically generated code that realizes the interactions between the subsystems. The timing constraints are realized by automatically generated code that embodies schedule and resource allocation decisions made by the supporting tools.

3.4 Rationale

There are two types of rationale information associated with a PSDL architecture: component specifications and requirements links.

Component specifications describe the required behavior for the components that can fit into a slot in the architecture. These specifications can be used as queries against a software base to automatically search for reusable components that can fill the slot. The specifications also serve to document the requirements of the slot as well as the degrees of freedom that the architecture allows for filling the slot. These requirements identify the properties of a component that should be tested or proved to ensure that it can reliably satisfy the requirements of a given slot in the architecture. The Spec language provides a formal notation for component specifications that

can express partial constraints on component behavior and generic parameterized specifications as well as completely determined specifications for individual components.

Requirements links connect parts of the architecture to higher level requirements that motivated the parts. All aspects of the architecture description can have requirements links, including individual components, connections, and constraints. The requirements are named entities that represent goals of the stakeholders of the system. They are represented informally, in terms that the stakeholders of the system understand. The requirements links support user reviews of an architecture as well as tools for supporting the evolution of the architecture.

3.5 Hierarchy

In PSDL, a software architecture is a hierarchical structure that can be represented as an operator realization tree and a type realization graph.

The root of the operator realization tree represents the entire system (the software and all external systems that interact with the software). Each operator in the tree is labeled with a connection pattern that defines its architecture, as illustrated in Fig. 2. The children of each operator in the tree are the component operators that appear in its connection pattern. The atomic operators at the leaves of the tree have empty connection patterns.

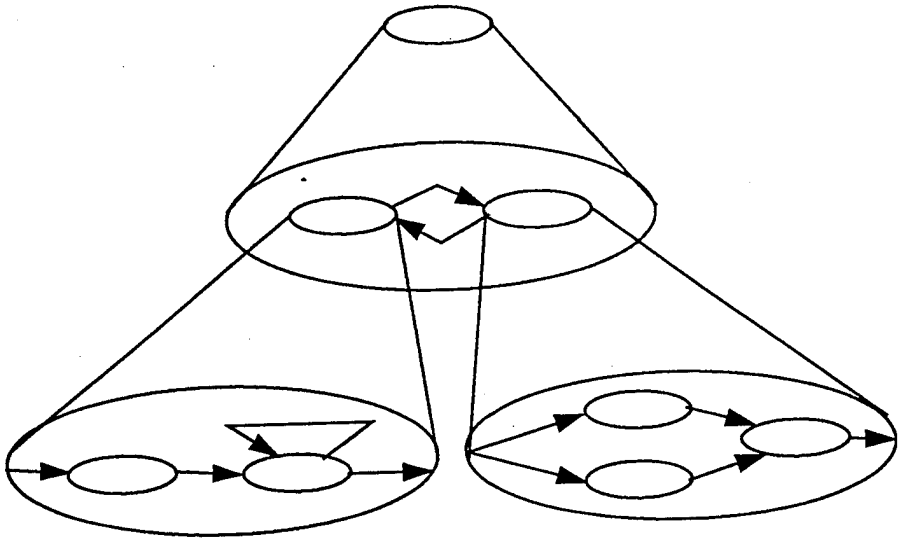

Figure 2: A Hierarchical Architecture.

Implicit in the tree representation for operator realizations is the constraint that each operator can appear in the realization of at most one parent operator. This is not a serious constraint on the designer and does not preclude reuse within a system because distinct copies of an operator can appear in several different places in the operator realization tree.

The prohibition against sharing affects the behavior of operators with internal states, because distinct copies of a state machine have distinct and independent copies of the state. Thus state transitions of one copy do not affect the states of all the other copies of the machine. This

constraint helps to ensure that all interactions between subsystems are apparent and explicitly specified in their interfaces.

The type realization graph contains a node for every data type associated with a stream in the connection patterns. The children of each type in the graph are the other types used in its representation. Atomic types are terminal nodes in the graph. Each type is an abstract data type, and is associated with a set of operators that act on the instances of the type. Each of these operators is the root of its own operator realization tree.

Atomic components, both operators and types, are realized by program modules that conform to the component slots in the architecture.

3.6 Example

Architectures are most useful if they are easy to tailor with respect to some dimensions, while still providing some common properties along other dimensions. A partial description of a simple generic architecture for a secure communications channel is shown in Fig. 3.

This generic architecture can be tailored to provide different security properties by plugging in different versions of the encoder and decoder components. However, these two components are not independent. One of the consistency properties associated with the architecture is shown in Fig. 4. This constraint requires that the message will be transmitted without modification for properly matched keys, for all well-formed realizations of the architecture.

The detailed security properties of the communications channel are not completely determined by the architecture, and can vary with the choice of the encoder and decoder components. For example, a simple realization might require that two keys form a matched pair if and only if they are equal. In this case both keys must clearly be kept secret. A more sophisticated realization with public encryption keys might require that the encryption key be the product of two large prime numbers, and the matching private decryption key be one of the prime factors of the public key. A version with a higher level of security might require the relation between matched pairs of keys to depend on the time the message was sent.

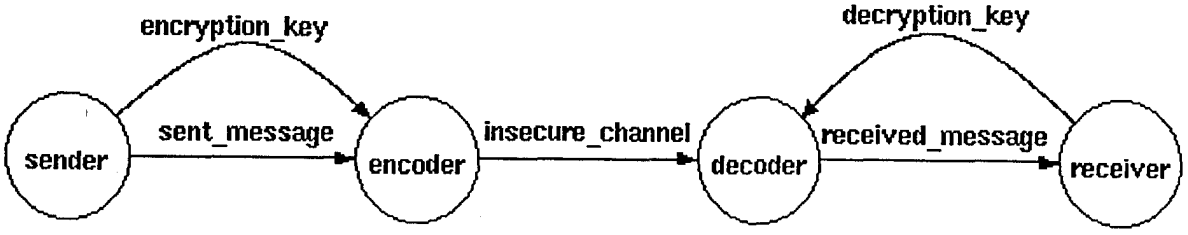

Figure 3: A Generic Secure Channel Architecture.

4 Automation support for software architectures

With respect to our focus on computer-aided prototyping, the most basic kind of support for software architectures is generating code for realizing the specified interactions between the com-

```
OPERATOR encoder
  SPECIFICATION
    INPUT
      encryption_key : key,
      sent_message : message
    OUTPUT
      insecure_channel : message
  AXIOMS
    { ALL(m: message, k1 k2: key SUCH THAT matched-pair(k1, k2) ::
          decoder(k1, encoder(k2, m)) = m) }
END
```

Figure 4: An Instantiation Constraint of the Secure Channel Architecture.

ponents. This was one of the first capabilities developed for the CAPS system [4].

Another basic kind of automation support is using the architecture description to find components that can fill slots in the architecture. These components can be realized by individual modules of software or hardware, or by connection patterns given by a lower level architecture. We have developed a method that uses practical amounts of computational resources and can in some cases certify that an automatically retrieved component will meet the user's requirements, so that it can be used without inspecting the code [12]. This method requires component specifications to be associated with the reusable components in the software base as well as with the component slots in the software architecture. There is a tradeoff between computational effort and the proportion of behavioral matches that can be detected by such a system. Completeness is unattainable because exact specification matching is undecidable for specifications with sufficient expressive power to represent the full range of designer intentions. Work is currently under way to explore related methods and representations for component behavior that are easier to use, to make this capability accessible to a larger group of software developers.

More recent efforts use architectural information to provide decision support for software evolution. This part of the effort uses a graph model to represent the derivation history of the architecture [5]. The graph contains different versions of the architectural information for the system, including hierarchies of connection patterns for the subsystems and requirements. Dependencies between the components are captured by representations of design steps that are linked to the versions of the architectural description units they produce and to the versions of other architectural description units that were used to derive the new version. This structure can represent parallel and reconverging threads of development as well as the more common linear chains of development steps. The structure contains proposed and planned steps in addition to completed steps and their products. Several kinds of support for software evolution and team coordination are based on this information.

The evolution control system of CAPS [2] uses the requirements dependency information together with the structure of the previous version of the architecture to derive an approximate

work breakdown structure for responding to a proposed requirements change. The project manager approves proposed changes, and adjusts the approximate work plans to account for possible new subsystems required by the change and to identify modules that do not change even though they contribute to the changed requirements. The manager also adds effort estimates and policy information such as priorities of different changes, deadlines, and skill requirements for different designer tasks. The system then uses information about the team of available designers to project a schedule, assign tasks to designers when they become free, and automatically monitor the progress of the project against the deadlines. It also automates configuration management based on the work plan by automatically checking out the proper versions of the documents needed to complete a design task, putting them into the responsible designer's private work space, and automatically checking the results back into the repository with the proper dependencies and version identifiers when the task is done and associated checking procedures have succeeded.

CAPS also has facilities for automatically combining the effects of two changes to an architecture, and for checking the semantic consistency of the two changes [8]. This facility is particularly useful when different aspects of an architecture are undergoing concurrent exploratory development.

5 Extensions

PSDL has a relatively static view of software architectures because it supports automatic methods for realizing hard real-time constraints on system behavior. The number of time-critical functions must be bounded to guarantee timing constraints can be met with fixed computational resources. This is achieved in PSDL by requiring that the set of components be statically declared. This implies that dynamic reconfiguration of software architectures is limited in the PSDL model: all possible components and connections must be statically declared, and resources must be allocated for the worst case. Reconfiguration in this context amounts to dynamically enabling or disabling selected connections and components from the fixed set of possibilities. Such reconfiguration is readily expressible via PSDL control constraints and a set of data streams carrying descriptions of the current system configuration.

This situation can be extended in several ways.

- Dynamic component creation can be allowed for components and connections that do not have any timing constraints. In such a case system response times can increase with the number of currently active instances of components in a dynamic software architecture.

- Dynamic component creation may be tractable for time-critical components if component creation is linked to creation of additional processors and connections. This model makes sense for large distributed systems with long lifetimes, where new hardware nodes can be added while the system is in operation. For example, it is entirely reasonable to assume that each new airplane will have its own set of onboard computers. However, limitations on the communications bandwidth and network diameter (maximum path length) still seem to require bounds on the maximum size of the dynamic configuration of such an architecture to guarantee service within a fixed deadline.

A challenge that arises in the description of dynamic architectures is developing understandable representations of the connection rules. For static systems a connection graph presents a convenient and understandable representation of component interactions that can be displayed and edited graphically. For dynamic systems more abstract representations are needed to capture the parts of the interaction protocols that remain invariant across the dynamic reconfigurations. Fully general solutions appear to require powerful notations that can be hard to read and can require high skill levels to use.

Readily understandable representations are possible if we constrain the degrees of freedom in the dynamic architecture. A simple example of a tractable and constrained kind of dynamic architecture is one that allows a variable number of instances of the same generic component that share a common role in a given interconnection pattern. This construction can be used for time-critical operations if the number of instances in the collection is bounded. An example of a possible graphical representation for such a connection pattern is shown in Fig. 5. The example is a fragment of an architecture for a display system that supports multiple windows.

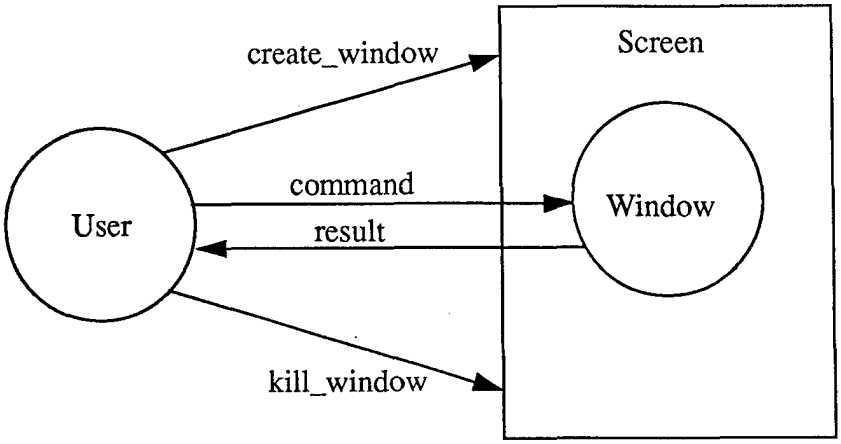

Figure 5: A Dynamic Architecture.

This example illustrates an extension to PSDL that supports dynamic collections of components as architectural building blocks. The component shown inside the collection box is a generic template that can have zero or more instances. In the example, the screen is a collection of windows. Creation of new windows and destruction of windows are operations performed by the collection; the streams create_window and kill_window are therefore directed to the collection rather than to the instances. These streams carry values of type window_id, which correspond to the generic parameter of the window template and serve to identify the instances of the collection to be added or removed.

The default communications pattern into such a collection is broadcast to all of the instances. The use of generic parameters to distinguish among the instances of the replicated component enables PSDL control constraints to specify more selective message routing, to single out either individual instances or larger subsets of the collection.

The default communications pattern out of such a collection is writing into a common stream, which implicitly merges all of the communications into a linearly ordered sequence based on

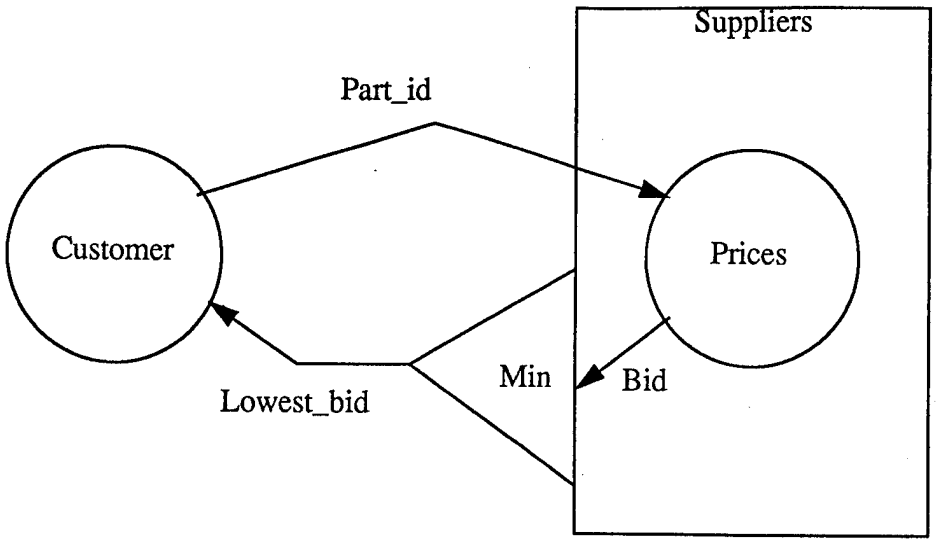

Figure 6: Reductions in A Dynamic Architecture.

writing time. The most common alternative to the default output pattern for collections is reduction, where all of the responses from the collection are combined using a summary function such as addition or maximum value.

We suggest supporting reduction patterns via an additional graphical primitive such as the one shown in figure 6. In this example suppliers is a collection of price databases that produce bids on orders from a customer. All of the bids in the collection are combined in a reduction that chooses the minimum value, resulting in the lowest bid. The interactions needed for price databases to enter and leave the collection are not shown; they have a structure similar to Fig. 5.

6 Conclusions

CAPS has been used to develop a variety of prototypes, including an architecture for a generic command and control station [6]. Our experience indicates that abstraction and generalization are essential for realizing the full benefits of systematic approaches to software architecture in the context of prototyping. The purpose of an architecture is to achieve system integration while preserving flexibility of system behavior. Human understanding plays an important role in the design of a software architecture, and simplicity of representation can help to use this scarce resource as effectively as possible.

Good software architectures should be able to accommodate most of the evolutionary changes to a software system and most of the alternative configurations in a system family with little or no impact on the architecture, by replacing selected components that fit into the architecture. This requires explicit design of the degrees of freedom to be supported by an architecture as well as the structures and constraints that enable an architecture to provide given types of capabilities and services. Thus the slots in an architecture should have general specifications that are compatible with many different components that can fit into the slot, while still ensuring that all components that fit will be able to work together in harmony to achieve the overall goal of the architecture.

Automated decision support is needed to make this work smoothly. Basic capabilities of a mature supporting environment include automated reuse and code generation capabilities for

both components and interconnections. More advanced capabilities include support for evolution of architectures, design team coordination, and analysis and testing support for both components and connection patterns to create effective architectures and to assess their effectiveness.

References

[1] Special issue on software architectures, *IEEE Trans. on Software Eng. 21*, 4, (April 1995).

[2] S. Badr and Luqi, Automation Support for Concurrent Software Engineering, *Proc. of the Sixth International Conference on Software Engineering and Knowledge Engineering*, Jurmala, Latvia, June 20-23, 1994, pp. 46–53.

[3] V. Berzins and Luqi, *Software Engineering with Abstractions*, Addison-Wesley, 1991, ISBN 0-201-08004-4.

[4] Luqi, Automated Prototyping and Data Translation, *Journal of Data and Knowledge Engineering 5*, (July 1990), pp. 167–177.

[5] Luqi, A Graph Model for Software Evolution, *IEEE Trans. on Software Eng. 16*, 8, (Aug. 1990), pp. 917–927.

[6] Luqi, Computer-Aided Prototyping for a Command-and-Control System Using CAPS, *IEEE Software 9*, 1, (Jan. 1992), pp. 56–67.

[7] Proc. of Monterey Workshop on Increasing the Practical Impact of Formal Methods for Computer-Aided Software Development: Software Evolution Monterey, CA, Sep. 1994.

[8] Special Issue on CAPS, *Journal of Systems Integration 6*, 1/2, (1995).

[9] W. Myers, Taligent's CommonPoint: The Promise of Objects, *Computer 28*, 3 (March 1995), pp. 78–83.

[10] D. Parnas, On the Criteria to be Used in Decomposing a System into Modules, *CACM 15*, 12 Dec. 1972, pp. 1053–1058.

[11] B. Ramesh and Luqi, Process Knowledge Based Rapid Prototyping for Requirements Engineering, *Proceedings of IEEE/ACM Symposium on Requirements Engineering*, San Diego, CA, Jan. 1993, pp. 248–255.

[12] R. Steigerwald, Luqi and J. McDowell, CASE Tool for Reusable Software Component Storage and Retrieval in Rapid Prototyping, *Information and Software Technology 33*, 9, Nov. 1991, pp. 698–706.

[13] G. Wiederhold, P. Wegner, S. Ceri Toward Megaprogramming, *CAM 35*, 11, (Nov. 1992), pp. 89–99.

Parameterized Programming and Software Architecture*

Joseph A. Goguen
Programming Research Group, Oxford University Computing Lab

Abstract: This paper discusses an approach to software architecture based on concepts from parameterized programming, particularly its language of "module expressions." A module expression describes the architecture of a system as an interconnection of component modules, and executing the expression actually builds the system. Language features include: modules parameterized by theories, which declare interfaces; a number of module composition operations; views for binding modules to interfaces; and both vertical and horizontal composition. Modules may involve information hiding, theories may declare semantic restrictions with axioms, and views assert behavioral satisfaction of axioms by a module. Some "Laws of Software Composition" are given, showing how various module composition operations are related. The paper also shows how a variety of communication styles can be supported in this approach, and how it can be extended to provide support for evolution and traceability. All this is intended to ease the development of large systems, and in particular, to make reuse more effective in practice.

1 Introduction

This paper presents an approach to software architecture that is based on concepts from parameterized programming. Parameterized programming [5, 6] concerns design and module composition times, rather than compile or run times; this is, it addresses the architectural level of software. It supports building systems, software reuse, and controlled evolution, as well as the management of configurations, versions, families, documentation, etc. The "module expressions" used in parameterized programming constitute a module connection language (abbreviated MCL, and sometimes also called an architecture description language, or ADL). A module composition language may be used

1. *descriptively*, to specify and analyze given design, or
2. *constructively*, to describe a new design using existing modules, and execute it to build a new system.

Detailed design and coding are unnecessary for construction or description if a suitable database (hereafter called a library) of specifications and relationships among them is available. Parameterized programming supports both construction and description, assuming availability of a library that contains:

1. **module expressions**, describing systems as interconnections of modules, and
2. a **module graph**, describing modules and relationships among them.

*The research reported in this paper has been supported in part by the Office of Naval Research, the Ada Joint Project Agency, ARPA as part of its DSSA (Domain Specific Software Architecture) project, the UK Science and Engineering Research Council, the CEC under ESPRIT-2 BRA Working Groups 6071, IS-CORE (Information Systems COrrectness and REusability), and 6112, COMPASS (COMPrehensive Algebraic Approach to System Specification and development), Fujitsu Laboratories Limited, and the Information Technology Promotion Agency, Japan, as part of the R and D of Basic Technology for Future Industries "New Models for Software Architecture" project sponsored by NEDO (New Energy and Industrial Technology Development Organization).

A module graph can incorporate executable code for modules, as well as specifications, and other software objects. When a module expression is executed in the presence of a suitable module graph, an executable system can be constructed by manipulating and linking the implementation modules.

The parameterized programming approach to software architecture has been validated by experience with LILEANNA [21], an MCL using Ada for implementation and Anna [15, 16] for specification. LILEANNA was developed as part of the DSSA ADAGE project sponsored by ARPA. The implementation was done by Will Tracz of Loral Federal Systems, and has been used for helicoptor navigation software. This approach seems especially useful for "software factory" situations, such as the Loral helicopter navigation software facility, where a number of similar systems are produced over time. In such cases, systems like LILEANNA may be able to save a great deal of software development time, although significant initial investment may be needed to accumulate information for the library.

LILEANNA has a formal semantics based on category theory, following ideas developed for the Clear specification language [3, 4]; more recently, a set theoretic semantics has been given [12]. These semantics are very general, and apply to languages other than Ada and Anna, and indeed, to any implementation-specification language pair that satisfies certain axioms. The properties of an interconnection of modules are related in a straightforward way to those of its components, because of the precise and straightforward semantics of module expressions. The work reported in this paper is based on ideas developed in 1983, and first reported in [5], which suggested a design for LIL, a library interconnection language for Ada; see also [6].

1.1 What is Architecture?

The term "architecture" has been much discussed in recent literature, and there is now a large family of partially overlapping definitions. This situation suggests that instead of arguing over the meaning a single popular term, we should develop new terminology that distinguishes among the most important concepts of software architecture. This subsection suggests one such terminology, following ideas from parameterized programming.

A narrow meaning of architecture concerns *static* aspects of systems, including structure, components, relationships among components, communication style, etc. A wider sense concerns the *dynamic* aspects of software development, such as the (ever changing) rationales for design choices, and their traceability back to the (ever changing) requirements (the importance of requirements and their evolution is further discussed in [8]). We may call these **static** and **dynamic architectures**, respectively. Some aspects of dynamic architecture are discussed in Section 5.

It is also important to distinguish the architecture of a particular system from the knowledge needed for constructing a family of related systems (or developing a large complex evolving system). We suggest calling these **system architecture** and **domain architecture**[1], respectively. In parameterized programming, a module expression captures a particular design, while a module graph captures an architectural domain. Both are needed to support large scale system development efforts.

Acknowledgements

I thank Will Tracz for many useful discussions, and for help with the examples in this paper.

[1]The word **framework** is sometimes used for a collection of modules capturing common aspects of applications over a certain problem domain.

2 Hyperprogramming and Module Graphs

Parameterized programming[2] assumes that all modules have associated specifications; these serve as "headers" for other information, particularly source code and compiled code. The specifications need not be complete, but must include at least the syntax of what is exported by the module, the syntax of its interface if it is parameterized, and the names of any modules that it imports. In parameterized programming, specification and code modules are collected together with other information to form a module graph, which describes the organization of the system development database, including information relevant to both system and domain architecture[3].

A node of a module graph may have as header a

- theory specification,
- package specification, or
- module expression,

while an edges of the module graph may be labelled with a module relationship, such as

- inheritance,
- view,
- parameterization, or
- instantiation.

Package specifications are intended to have associated executable code. Theory specifications are not, as discussed in the next subsection. Other software objects that maybe be associated with nodes or edges are also discussed below.

The organization of module graphs is based on some ideas from what we call **hyperprogramming** [7]. The most relevant ideas can be summarized as follows:

- modules are associated with **clusters**, where
- a specifications serve as *headers* for each cluster, and
- one or more implementations may be given for each package specification, including
  - source code, and
  - compiled code,
- plus (optionally)
  - test cases,
  - performance data,
  - documentation,
  - administrative information, such as programmer, date of last change, etc.,
  - rationale, and
  - links to other software objects.

Each cluster has its own node in a module graph. The other software objects referred to in the last point above might be requirements, dataflow diagrams, transcripts of interviews, review documents, etc.; these will also be attached to clusters associated with nodes in the module graph.

Here is a LILEANNA package specification for a stack of integers:

[2]The term *megaprogramming* is used for some rather similar ideas within the ARPA community [2, 22].

[3]The module graph is an *abstraction* of this organization. For various reasons, including efficiency and the structure of existing database systems, the information structure that is actually implemented may be quite different from that suggested by the mathematical structure of a graph.

```
package INTSTACK is
   import INTEGER;
   type Stack;
   exception Stack_Empty;
   exception Stack_Overflow;
   function Is_Empty(S: Stack) return Boolean;
   function Is_Full(S: Stack) return Boolean;
   function Push(I: Integer; S: Stack) return Stack;
   --| where
   --|       Is_Full(S) => raise Stack_Overflow;
   function Pop(S: Stack) return Stack;
   --| where
   --|       Is_Empty(S) => raise Stack_Empty;
   function Top(S: Stack) return Integer;
   --| where
   --|       Is_Empty(S) => raise Stack_Empty;
   --| axiom
   --|       for all I: Integer, S: Stack =>
   --|           if not Is_Full(S) then
   --|               Pop(Push(I, S)) = S and ;
   --|               Top(Push(I, S)) = I;
   --|           end if;
end INTSTACK;
```

Specifications need only be developed to the extent that it is practically useful to do so; often just the syntax is given. The above specification is incomplete, in that it does not define the functions Is_Full or Is_Empty.

2.1 Theories and Views

Theory specifications, called **theories** for short, are used to describe generic module interfaces; these may contain axioms, which serve as semantic constraints on the actual modules that are allowed by the interface. There are no implementation modules associated with theories.

The simplest theory just says that a type should be provided:

```
theory TRIV is
   type Element;
end TRIV;
```

Any (non-empty) module can be matched with TRIV, by designating one of its types.

The next theory has some axioms that provide semantic constraints on what can fit the interface that it defines, saying that the type should have a partial order structure:

```
theory POSET is
   type Element;
   function <= (X,Y: Element) return Element;
   --| axiom
   --|   for all X,Y,Z: Element =>
   --|     X <= X and;
   --|     if X <= Y then Y <= X end if; and;
   --|     if X <= Y and Y <= Z then
   --|       X <= Z end if;
end POSET;
```

Views are used to bind actual modules to generic module interfaces, in order to instantiate generics. Because modules may involve information hiding, views only assert the *behavioral* sat-

isfaction of the axioms in their source theory by the target module[4]. However, *proving* that such axioms are satisfied should not be part of a system intended to be practical for ordinary use. Instead, views are used for recording a programmer's belief that the axioms in the source theory hold in the target module. Support for such a belief may be provided off line on the back of an envelope, by giving a formal mechanical proof, or by anything between these extremes; the belief may even be left unsupported, although this is not recommended. The nature of the support given (e.g., a scanned image of the envelope back, or the source file of the machine proof) can be stored with the view in the module graph.

The view below asserts that the relation => on integers satisfies the axioms of POSET; it can be used to instantiate the generic LILEANNA package SORT given later, yielding a package with a function that sorts integers in *descending* order.

```
view GEQ :: POSET => Standard is
   types (Element => Integer);
   ops ("<=" => ">=");
end GEQ;
```

Another use of views is to assert global properties of systems, by giving a view to a top level module (represented by a module expression saying how the system is composed from lower level modules) from a theory with axioms giving the properties to be asserted. As before, these axioms need only be behaviorally satisfied.

3 Parameterized Modules

The most characteristic feature of parameterized programming is the parameterization of modules over interface declarations: any module that "fits" the interface can be "substituted into" the parameterized module, yielding a new module that is an "instance" (or "instantiation") of the original module. The interface is described by a theory.

Below is LILEANNA code for a parameterized version of the INTSTACK module given earlier (material that is the same as in INTSTACK is indicated by .....):

```
generic package STACK[Element :: TRIV] is
   type Stack;
   .....
   function Push(E: Element; S: Stack) return Stack;
   .....
   function Top(S: Stack) return Element;
   .....
   -- | axiom
   -- |   for all E: Element, S: Stack =>
   -- | .....
end STACK;
```

The generic package specification below is for a sorting program, parameterized by the POSET theory, which says that a partially ordered set should be supplied in order for the program to work correctly:

```
generic package SORT[Item :: POSET] is
   importing LIST[Item];
   function Sort (X: List) return List;
   function Sorted (X,Y: List) return Bool;
```

[4]This means that the axioms need only appear to hold under all possible "experiments" on the module, involving its externally visible operations; sometimes this is also called *observational satisfaction.*

```
  -- | axiom
  -- |   for all X: List =>
  -- |     Sorted(Sort(X)) = true and;
  -- |     ..... ;
end SORT;
```

3.1 Instantiation

The SORT program specified above can be instantiated with the view GEQ simply by writing SORT[GEQ]; this results in a program for sorting integers in descending order. Similarly, we could instantiate SORT with the partial ordering relation of divisibility on natural numbers by writing SORT[DIV], where DIV is a suitable view.

Default views enable "obvious" views to be replaced by the name of their target module, or even the name of a type. For example, we would not have to write out a view from POSET to Standard that mapped the type Elt to Integer and the operation symbol => to itself, but could just write SORT[Integer]. Default views are computed using a certain set of default rules that are given in [6], and that capture many of our intuitions about what is "obvious". The following are some instantiations that use default views:

```
STACK[Integer]
STACK[LIST[Integer]]
STACK[STACK[Float]] .
```

(Default views were first implemented in OBJ3 [13], and are partially implemented in LILEANNA.)

An interface theory can call for more than one operation, and more than one type, and views can be written that express bindings to such interfaces. Generic modules can also have more than one interface, each defined by its own theory. These will require more than one view for instantiation.

3.2 Vertical Composition

Vertical structure describes the use of lower layers (virtual machines), whereas *horizontal* structure describes a given layer; the distinction between vertical and horizontal structure was first named and formalised by Goguen and Burstall [9]. Parameterized programming provides parameterization and instantiation for both vertical and horizontal structure. The following generic package specification has one horizontal and one vertical parameter:

```
generic package SORT[Item :: POSET](LISTP :: LIST[Item]) is
  function Sort (X: List) return List;
  function Sorted (X,Y: List) return Bool;
  -- | axiom
  -- |   for all X: List =>
  -- |     Sorted(Sort(X)) = true and;
  -- |     ..... ;
end SORT;
```

Note that the horizontal interface theory is itself parameterized, and moreover is instantiated with the horizontal formal parameter. There is also a default view from TRIV to POSET involved in the vertical instantiation, since SORT wants a POSET whereas LIST wants a TRIV. Also note that the notation [_] is used for horizontal parameterization, while (_) is used for veritcal parameterization. The same conventions apply to instantiation, as illustrated by the following module expression (where each instantiation uses a default view):

```
SORT[Natural](LIST7[Natural]) .
```

Modules can also import (or "inherit") other modules. The simple syntax for this is illustrated in the following:

```
package M42 is
    inherits SYS31;
    inherits Sort[Integer];
      ..... ;
end M42;
```

Inherited submodules are always shared, that is, new copies are not produced.

Figure 1 is a graphical evocation of the relationships among inheritance, horizontal parameterization, and vertical parameterization. Note that under horizontal instantiation, actual parameters are shared, whereas under vertical, new copies are used. Here M is a module, I is an imported module, T_H is a horizontal interface theory, and T_V is a vertical interface theory.

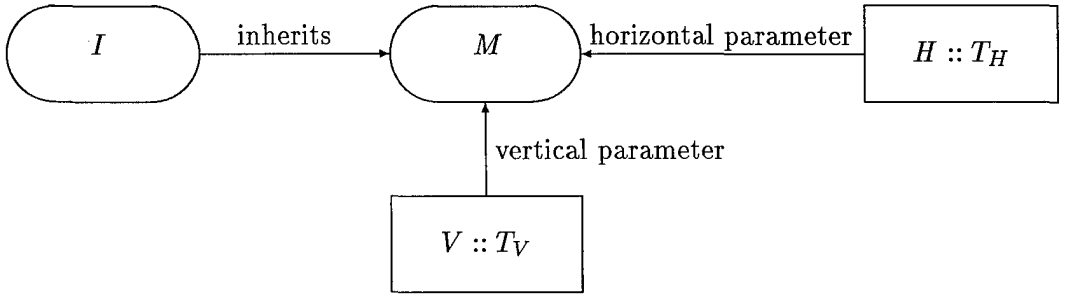

Figure 1: Horizontal and Vertical Composition

4 Module Expressions

The purpose of module expressions is to define software architectures. Therefore module expressions need more than parameterization and instantiation. The following additional operations for combining modules are implemented in LILEANNA:

- module aggregation, i.e., simple combination, taking account of
  - shared inherited modules, and
  - parameterization;
- deleting functionality;
- modifying functionality; and
- adding functionality.

Both operations and types can be renamed, deleted or added, as can exceptions, axioms, etc. And of course, both horizontal and vertical instantiation are allowed. The following module expressions illustrate the syntax:

```
SYS1 + SYS2
SYS1 * (rename op Put => Write) + SYS2 * (delete op Undo)
STACK * (rename type Stack => Stack1)[Integer]
  + STACK * (rename type Stack => Stack1)[LIST[Float]]
```

Thus + "adds" modules, i.e., forms a system containing all the summands; because summation is associative (see Section 4.2), any number of summands can be given without using parantheses. See [1] for some related work.

The **make** command "executes" a module expression to actually build a (sub)system, give it a name, and store it in the environment. The following illustrate this:

make SYS3 **is** STACK[STACK[Float] + SYS2 * (**delete op** Undo) **end** SYS3;

A **make** can also be parameterized, as in the following:

make SYS4[X :: POSET] **is** SORT[X](LIST12[X]) **end** SYS4;

Of course, a make command can use results of previous makes, as in

make SYS5 **is** SYS3 + SYS4[Float] **end** SYS5;

Furthermore, a **make** statement can be used to impose sharing of parameters on its constituents, as in the following:

make M329[X :: POSET] **is** SORT[X](LIST12[X]) + STACK[X] **end** M329;

Whenever a module expression is introduced, it is added as a node to the module graph, and whenever a module expression is evaluated, executable code is generated and attached to the cluster of the module expression.

Module expression evaluation can be implemented by manipulating intermediate compiled code (for LILEANNA, this is Ada's intermediate compiled code language DIANA). Intermediate compiled code is easier to manipulate than object code, while source code does not have much of the required information in a sufficiently explicit form. One also gets the benefit of being able to put the manipulated code through the compiler's backend, including optimization. Module expressions were first implemented in the OBJ system [13], but using different techniques, because there is no attached code in a separate programming language. The mathematical semantics of module expression evaluation is given by the colimit of a diagram extracted from the module graph [4].

LILEANNA provides a graphical "layout editor" for module expressions, based on notation like that typically used by engineers, i.e., boxes and arrows. This helps users to edit existing module expressions to define new (sub)systems, or define them from scratch, as they wish.

4.1 Architectural Styles

Many different architectural styles can be supported by parameterized programming, including different communication styles, such as shared variables, pipelining, message passing, and blackboarding. These can be described by using shared submodules in various ways. (Such a cell is a simple "object" in the sense of object oriented programming, and can be easily specified in LILEANNA.) For example, if a "cell" module C encapsulates a variable X, and if modules F and G each inherit C, then they share X, and in the system F + G, they can communicate through it. Similarly, a "post office" module can be inherited by a set of modules, and then used for passing messages among them.

Pipelining is a special case of shared variable communication, where only two modules share each cell; the cells represent the pipes, with one of the importing module reading the variable, and the other one writing it. For example, if modules F and G import a cell C1 and modules G and H import a cell C2, then F + G + H can function as a (short) pipeline.

The following subsection sketches ways to describe avionics architectures using module expressions. Here the modules encapsulate various digital signal processing subsystems, and shared cells represent "wires" that pass the signals.

4.1.1 Describing Avionics Systems with Parameterised Programming

An interesting example in this domain of a module that takes other modules as parameters is a Kalman filter module K that needs a model A of the aircraft; this situation is expressed by the simple module expression K[A]. It makes sense to parameterize K by aircraft models, so that the same filter can be used for many different models.

Now consider a flight control system F that needs a guidance system G, where G needs an aircomputer A. We can describe this situation with the module expression F[G[A]]. (We omit details of the code here and hereafter.) Assuming that the modules F and G have parameter (interface) theories GS and AC respectively, then views are needed to match A to AC and G to GS; it is natural to expect that default views will work in such examples.

This simple approach based on instantiating parameterized modules works well if the system architecture is linear, or more generally, a tree; but it is not adequate for sharing among more than two modules, or for feedback loops. Feedback is necessary in avionics software, because feedback control is a crucial technique.

For example, suppose the value of a floating point variable X in A needs to be fedback into F. We can capture this by encapsulating X in C, and letting F and A each inherit C; this system is still described by the simple module expression F[G[A]]. Another approach is to provide each F and A with a new parameter for a floating point cell, in which case the system is described by the module expression F[G[A[C]], C].

This example highlights the importance of sharing for a module interconnect formalism. We have shown that parameterised programming can accomplish such sharing in several ways. Without such a capability, it would be difficult or impossible to handle feedback and variables shared among several modules.

4.2 Some Software Laws

There are many relationships among the various operations on modules; the more important of these can be considered "laws of software." Here are a few of them:

$$M + M' = M' + M \ ,$$

$$M + (M' + M'') = (M + M') + M'' \ ,$$

$$M + M = M \ ,$$

$$M + N = M \text{ if } N \text{ inherits } M \ ,$$

$$M^H + M'^H = (M + M')^H \ ,$$

$$M^{N_1,N_2} = M^{N_2,N_1} = M^{N_1+N_2} \ ,$$

$$M^{N_1^H,\ N_2^H} = M^{(N_1+N_2)^H} \ ,$$

$$M^N + N = M^N \ ,$$

where M^N indicates that M inherits the module N, and M^H indicates that M inherits a set H of modules. There are many more laws, e.g., for parameterization; see [12] for these, as well as for proofs. These laws are used in the implementation of LILEANNA for simplifying module expressions.

5 Support for Evolution and Traceability

A traditional view is that software evolution only occurs after initial development is complete. By contrast, we consider evolution to include all activities that change a system, as well as the relationships among those activities, occurring throughout the system's life. Thus, the term "evolution" focuses attention on *change*, which is inevitable and unending throughout software development. Moreover, since large complex systems are inevitably embedded in complex evolving social contexts, they will necessarily *co-evolve* with those contexts, in the sense that each will affect the evolution of the other. (See [10] for further discussion of change and social context.)

The ubiquity of change motivates the use of iterative lifecycle processes, and especially prototyping, i.e., quickly building and evaluating a series of prototypes, which are concrete executable models of selected aspects of a system [17]. The ability of parameterized programming to describe software architectures, in both the domain and systems senses, can greatly facilitate prototyping. In some cases, all that need be done is edit a module expression. In other cases, the module graph may need updating, e.g., writing new modules or modifying old ones. Then executing the module expression yields a running prototype, which can be gracefully evolved into the actual system, and thereafter further evolved.

The additional information needed to cope with the dynamic evolution of (families of) software systems is provided by enriching the module graph with relevant relationships among various software objects, such as requirements and rationales. Rationales should be considered part of evolution support rather than architecture, because they must evolve along with the objects that they concern.

5.1 Traceability

The Centre for Requirements and Foundations at Oxford has projects to improve the acquisition, traceability, accessibility, modularity, and reusability of the numerous objects that arise and are manipulated during software development, with a particular focus on the role of requirements. An initial study [14] administered a detailed two-stage questionnaire to software engineers at a large firm, and found that traceability was considered the most important outstanding problem. Further analysis showed that there are actually several different traceability problems, which should be treated in different ways. Major distinctions are between pre-RS (Requirements Specification) traceability and post-RS traceability, and between forward and backward traceability.

Tracing back as far as requirements is important for developing large software systems. But it is also difficult because of the overhead of maintaining the huge mass of dependencies among the many objects produced by a large software development effort. Moreover, dependencies reaching far across the development cycle can be significant. Without formal representations for the objects involved, formal models for the dependencies, and tool support for managing them, it can be impossible to know what effect a change will have, and in particular, to know what other objects may have to be changed to maintain consistency. A hypergraph model for maintaining evolving dependencies, suitable as a basis for tool development, is given in [18]; this structure can be applied to give a model for evolving module graphs. Work on capturing domain knowledge has used methods from sociology, particularly ethnomethodology and its disciplines of conversastion and interaction analyses. See [8, 10] for further information on this research.

We are also developing a system called TOOR to support tracing dependencies among evolving objects, and in particular, to show how decisions are grounded in prior objects [19]. Significant subproblems include formalizing dependencies, and developing methods for calculating dependencies and for propagating the implications of changes. This approach, called *hyperrequirements*, builds on parameterized programming and hyperprogrammming, and is intended to support the social context of decisions, as well as their traceability, by linking related objects, based on the broad view of context and requirements suggested in [8].

TOOR supports user-definable relations, to allow differentiating among different links between the same objects. It also allows declaring mathematical properties of relations, such as transitivity, to give additional power and flexibility in tracing links through specifiable compositions of relations. TOOR supports several different trace modes, including browsing and regular expression based search, and it also supports module definition through an intuitive template-driven graphical interface. Hyperprogramming and hyperrequirements support reuse, through the generalized

notion of relation for linking objects for requirements, design, specification, coding, documentation, maintenance, etc.

TOOR is built on FOOPS [11, 20], a general object oriented language with specification capabilities. This makes TOOR particularly suitable for use in an object oriented development paradigm. TOOR uses FOOPS-like modules to declare software objects and relations, and to automatically create links as objects are interconnected and evolve. TOOR also provides hypermedia facilities, based on HTML, to make its use closer to analysts' intuitions and natural activities. For example, graphs, charts, and videos can be linked, as well as conventional documents.

6 Discussion and Conclusions

We have seen that parameterized programming can provide a systematic approach to software architecture, through its use of theories and views, its powerful methods for combining and modifying components to form new systems, and its underlying module graph data structure. The notions of module expression and module graph, we believe, add some clarity to discussions about the nature of software architecture. We have illustrated the use of module expressions to achieve several different architectural styles.

Parameterized programming extends to hyperprogramming and hyperrequirements, to support evolution by including other software objects in the module graph, such as requirements, rationales, and documentation, as well as relations to support traceability. This enables design objects and relations to be managed in a systematic way, which can have a significant impact on the reusability of code, through the reuse of design information. Support for evolution also means that prototyping and traceability integrate with the approach in a natural way. Development system integration and enhanced reusability are the result of a systematic way of storing information in an evolving module graph. It is hoped that ideas like these will make the development of large complex systems more reliable and efficient.

References

[1] Don Batory and Sean O'Malley. The design and implementation of hierarchical software systems with reusable components. Technical Report TR–91–22, Department of Computer Sciences, University of Texas, Austin, 1991. Revised May 1992.

[2] Barry W. Boehm and William L. Scherlis. Megaprogramming. In *Proceedings of Software Technology Conference 1992*, pages 63–82, April 1992.

[3] Rod Burstall and Joseph Goguen. Putting theories together to make specifications. In Raj Reddy, editor, *Proceedings, Fifth International Joint Conference on Artificial Intelligence*, pages 1045–1058. Department of Computer Science, Carnegie-Mellon University, 1977.

[4] Rod Burstall and Joseph Goguen. The semantics of Clear, a specification language. In Dines Bjorner, editor, *Proceedings, 1979 Copenhagen Winter School on Abstract Software Specification*, pages 292–332. Springer, 1980. Lecture Notes in Computer Science, Volume 86; based on unpublished notes handed out at the Symposium on Algebra and Applications, Stefan Banach Center, Warsaw, Poland, 1978.

[5] Joseph Goguen. Suggestions for using and organizing libraries in software development. In Steven Kartashev and Svetlana Kartashev, editors, *Proceedings, First International Confer-*

ence on Supercomputing Systems, pages 349–360. IEEE Computer Society, 1985. Also in *Supercomputing Systems*, Steven and Svetlana Kartashev, Eds., Elsevier, 1986.

[6] Joseph Goguen. Principles of parameterized programming. In Ted Biggerstaff and Alan Perlis, editors, *Software Reusability, Volume I: Concepts and Models*, pages 159–225. Addison Wesley, 1989.

[7] Joseph Goguen. Hyperprogramming: A formal approach to software environments. In *Proceedings, Symposium on Formal Approaches to Software Environment Technology.* Joint System Development Corporation, Tokyo, Japan, January 1990.

[8] Joseph Goguen. Requirements engineering as the reconciliation of social and technical issues. In Marina Jirotka and Joseph Goguen, editors, *Requirements Engineering: Social and Technical Issues*, pages 165–200. Academic Press, 1994.

[9] Joseph Goguen and Rod Burstall. CAT, a system for the structured elaboration of correct programs from structured specifications. Technical Report Report CSL-118, SRI Computer Science Lab, October 1980.

[10] Joseph Goguen and Luqi. Formal methods and social context in software development. In Peter Mosses, Mogens Nielsen, and Michael Schwartzbach, editors, *Proceedings, Sixth International Joint Conference on Theory and Practice of Software Development (TAPSOFT 95)*, pages 62–81. Springer, 1995. Lecture Notes in Computer Science, Volume 915.

[11] Joseph Goguen and José Meseguer. Unifying functional, object-oriented and relational programming, with logical semantics. In Bruce Shriver and Peter Wegner, editors, *Research Directions in Object-Oriented Programming*, pages 417–477. MIT, 1987. Preliminary version in *SIGPLAN Notices*, Volume 21, Number 10, pages 153–162, October 1986.

[12] Joseph Goguen and Will Tracz. An implementation-oriented semantics for module composition, 1995. In preparation.

[13] Joseph Goguen, Timothy Winkler, José Meseguer, Kokichi Futatsugi, and Jean-Pierre Jouannaud. Introducing OBJ. In Joseph Goguen and Grant Malcolm, editors, *Algebraic Specification with OBJ: An Introduction with Case Studies.* Cambridge, to appear. Also Technical Report, SRI International.

[14] Orlena Gotel. Requirements traceability. Technical report, Centre for Requirements and Foundations, Oxford University Computing Lab, December 1992.

[15] Bernd Krieg-Brückner and David Luckham. ANNA: Towards a language for annotating Ada programs. *SIGPLAN Notices*, 15(11):128–138, November 1980.

[16] David Luckham. *Programming with Specifications: An Introduction to* ANNA, *A Language for Annotating Ada Programs.* Springer, 1990.

[17] Luqi. Software evolution through rapid prototyping. *IEEE Computer*, 22(5):13–25, 1989.

[18] Luqi and Joseph Goguen. Formal methods: Problems and promises. *IEEE Software*, 1996. To appear.

[19] Francisco Pinheiro and Joseph Goguen. Design and use of an object-oriented tool for tracing requirements. *IEEE Software*, to appear, March 1996. Special issue of papers from ICRE '96.

[20] Adolfo Socorro. *Design, Implementation, and Evaluation of a Declarative Object Oriented Language.* PhD thesis, Programming Research Group, Oxford University, 1994.

[21] Will Tracz. Parameterized programming in LILEANNA. In *Proceedings, Second International Workshop on Software Reuse*, March 1993. Lucca, Italy.

[22] Gio Wiederhold, Peter Wegner, and Stefano Ceri. Toward megaprogramming. *Communications of the ACM*, 35(11):89–99, 1992.